Unistack: An Interoperable Runtime Environment for Exascale Systems

Pavan Balaji (Argonne) Paul Hargrove (Berkeley) Laxmikant Kale (UIUC)
Sriram Krishnamoorthy (PNNL)

1 Introduction

Computing research has produced a rich variety of mature parallel programming models and their associated runtime systems, each with its individual set of capabilities and advantages demonstrated on DOE applications. Different programming models provide differing idioms for expressing parallelism, managing the work of a computation, and operating on distributed and shared data structures. Their associated runtime systems provide the low-level capabilities needed to support these high-level models as well as a layer of abstraction needed to efficiently harness increasingly complex hardware systems. Message-passing programming models, such as MPI [10, 15], provide a portable mechanism to exchange data between pairs or groups of processes. Data space models, such as Global Arrays (GA) [17] and partitioned global address space (PGAS) languages [18], have addressed the requirements of applications by allowing them asynchronous access to globally distributed data. Compute or tasking models such as the MADNESS runtime [9, 11], ADLB [13], Scioto [7, 8], and Charm++ [12], on the other hand, have addressed the requirements of applications such as NAMD [2], NWChem [4], Green's function Monte Carlo (GFMC) [13], and MADNESS using load-balancing techniques, including work-stealing. While each programming model provides unique capabilities, the utility and potential for widespread adoption of each model is still limited by the inability of applications to use multiple models together. Despite the variety and broad availability of models and runtime systems, today's application developers must limit themselves to a single model in order to achieve reliable application behavior, performance, and portability.

As we move toward multi-petascale and exascale systems, both application requirements as well as hardware complexity are expected to continue to grow. Consequently, for applications to take full advantage of the massive parallelism of such systems, it is becoming increasingly clear that they need to combine the capabilities of multiple such models. A well-known example is an MPI application that needs to use a threading model within a node to better exploit the shared memory and many cores on the node. However, emerging applications and extreme-scale machines require a richer and more complex interleaving of programming models. For example, how can multimodule applications primarily based on Unified Parallel C (UPC) [18] or Coarray Fortran (CAF) [14] utilize math libraries written in MPI, such as PETSc [1], that have had hundreds of programmer-years of development invested in them? Similarly, how can an application written in GA utilize load-balancing tools written in Charm++? Can the threading infrastructures of Charm++ and OpenMP [5] coexist within the same application, over the limited resources available on the node? For promoting software reuse, especially for expensive HPC software, it is also important that one be able to reuse an existing module or library in a new application, irrespective of the programming model in which it is written.

2 The Unistack Runtime

What is required to enable applications to use multiple programming models simultaneously? In many cases, such capability either is not possible today or can be used only in an extremely restricted manner. For example, MPI and UPC, or UPC and GA, cannot naturally be used together today without special restrictions [6] because they use different runtime systems that are not aware of each other's existence: MPI has its own runtime system, UPC uses the Global Address Space Networking (GASNet) runtime system [3], and GA uses Aggregate Remote Memory Copy Interface (ARMCI) [16] internally. Accessing the same data object from both MPI and UPC typically leads to resource conflicts and data corruption. Similarly, CAF programs cannot utilize MPI libraries or Charm++ primitives without explicitly ensuring that their runtime systems do not conflict with each other; hence, the same data objects cannot currently be used in both models.

The goal of the Unistack approach is to define, design, and develop a natively interoperable runtime infrastructure that aims at allowing applications to use any combination of existing high-level programming models, including high-level global data space models, global compute space models, or even the low-level runtime infrastructure. Legacy applications written in MPI should be able to extend and implement newer features using other models. New applications written in upcoming programming models should be able to performance tune specific parts of their application by directly using low-level communication library capabilities or hardware topology information, without being concerned with interoperability issues. Applications should be able to use similar data management techniques for regular homogeneous systems, as well as accelerator-augmented heterogeneous systems. In short, Unistack aims at developing a framework that broadly involves a **unified** *low-level runtime infrastructure*, i.e., a unified software runtime that can simultaneously support multiple high-level programming models. Unistack comprises unification at two levels:

Unified Low-level Communication Libraries: Under the assumption that it has full control of the system, a communication library is often "selfish." It may block without yielding control to other software, leading to deadlock in many multimodel scenarios—a fundamental barrier to interoperability. Unification will ensure that progress is made on *all* outstanding communication at any point in the execution, removing a lack of *communication progress* as a barrier to interoperability. Selfish runtime practices include resource allocation, often approaching a platform's limits. Resources for transient use may be placed in private pools for reuse, rather than freed. For permanent resources, a doubling of usage to support two simultaneous models may exceed what is available. Such concerns become more acute as node counts increase and per core memory decreases. Unified resource management will reduce or remove *resource consumption* as a barrier to multimodel application development. Low-level communication runtime libraries are often optimized based on the narrow semantics of the programming model they support. One example is operations requiring memory allocated by (or registered with) the runtime. Unification of runtimes allows the same memory to meet requirements of multiple models, reducing or removing *usage constraints* as barriers to interoperability.

Unified Threading Runtime Systems: Most high-level programming models, including OpenMP, Charm++, and CAF, internally utilize programming-model-specific execution contexts (typically referred to as user-level threads) that are mapped onto threads exposed by the operating system or hardware. This approach allows these models to work around data dependency stalls within the application computation by scheduling a large number of user-level threads on a limited number of operating system (OS) threads. However, for multimodel applications that simultaneously utilize more than one programming model, this can cause their runtime systems to conflict with each other. Thus, the goal of the unified threading runtime is to provide a single interface for thread management that will be responsible for scheduling a variety of execution contexts across available resources and ensuring that the node-level workload is balanced. Multiple programming models can utilize the unified threading runtime for thread management, scheduling, and balancing load across threads. Unifying the scheduling engines of individual threading runtimes will allow all threads to be cooperatively scheduled in a manner that meets the application goals, rather than the goals of individual runtimes.

3 Summary

Challenges Addressed: Interoperability of existing and upcoming runtime systems.

Maturity: Several runtime systems have been funded and developed by DOE in the past few decades and have been widely used by a large number of key DOE applications. However, these runtime systems have been designed independently without any notion of "working together" so far. The time is right to take the lessons learnt from these mature projects and unify them into a single interoperable runtime system.

Uniqueness: While interoperability in itself is not unique to exascale, so far, most applications could get away with using a single programming model. However, as we move to exascale, this is no longer possible due to the increasing complexity of the applications as well as the architectures. For example, nuclear physics applications such as GFMC currently rely on a combination of message-passing and work-stealing models (based on ADLB), which has been shown to scale to tens of thousands of cores, enabling the study of the phenomena within the nucleus of an atom as complex as carbon-12. With increasing per task memory requirements of larger problems, however, using such a model alone without complementary global address space capabilities is impeding progress to larger elements, such as carbon-14 and oxygen-16. Such problems are also common in applications such as NWChem which utilizes global arrays for global data access. With the increasing hierarchy of processing elements, however, data access costs are becoming heavily dependent on the relative locality of the data and the processes accessing the data, making the task migration capabilities in tasking models more effective in some cases, compared with accessing data through global address space models.

Novelty: A unified runtime system is novel in its ability to simultaneously support multiple programming models—a feature that has been fundamentally missing in all existing runtime systems. This would be the first concentrated efforted to unify DOE investment in runtime systems for programming models into a single framework usable by applications.

Applicability: A unified runtime system for interoperable programming models is a key component that is impeding progress for many DOE applications.

Effort: A unified runtime infrastructure is a complex, but critical, undertaking that requires expertise in communication runtime layers, threading runtimes, and operating system support for light-weight communication and threads. A team of 4 FTEs working together for 3-5 years would be a reasonable estimate for this effort.

References

[1] S. Balay, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang. PETSc users manual. Technical report, Citeseer, 2004.

[2] J. A. Board, L. V. Kalé, K. Schulten, R. Skeel, and T. Schlick. Modeling biomolecules: Larger scales, longer durations. *IEEE Computational Science and Engineering*, 1(4), 1994.

[3] Dan Bonachea, Christian Bell, Paul Hargrove, and Mike Welcome. GASNet 2: An Alternative High-Performance Communication Interface, November 2004.

[4] E. J. Bylaska and et al. *NWChem, A Computational Chemistry Package for Parallel Computers, Version 5.1*, 2007.

[5] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. The MIT Press, 2007.

[6] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *Proceedings of the ACM International Conference on Computing Frontiers (CF)*, Bertinoro, Italy, May 2010.

[7] J. Dinan, S. Krishnamoorthy, D. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, 2008.

[8] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[9] G I Fann, R J Harrison, G Beylkin, J Jia, R Hartman-Baker, W A Shelton, and S Sugiki. MADNESS applied to density functional theory in chemistry and nuclear physics. *Journal of Physics: Conference Series*, 78(1):012018, 2007.

[10] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994.

[11] R. J Harrison. Multiresolution ADaptive NumErical Scientific Simulation (MADNESS), 2010. see http://www.csm.ornl.gov/ccsg/html/projects/madness.html.

[12] L. V. Kale and S. Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, New York, NY, USA, 1993. ACM.

[13] E. Lusk, S. Pieper, and R. Butler. More SCALABILITY, Less PAIN. *SciDAC Review*, (17):30–37, 2010.

[14] J. Mellor-Crummey, L. Adhianto, W. Scherer III, and G. Jin. A new vision for Coarray Fortran. In *PGAS '09: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, pages 1–9, NY, NY, USA, 2009. ACM.

[15] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[16] J. Nieplocha and M. Krishnan. High performance remote memory access comunications: The ARMCI approach. *International Journal of High Performance Computing and Applications*, 20:2006, 2005.

[17] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[18] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.